MODELING AND PERFORMANCE EVALUATION

Salvatore Orlando

Basics

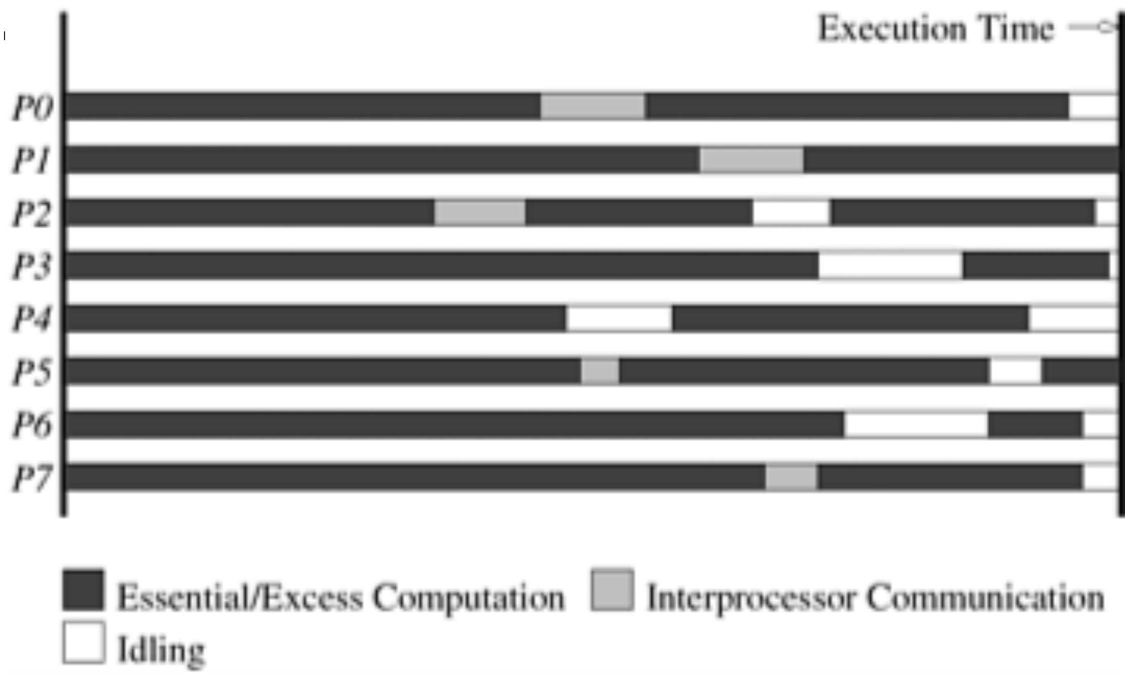
- **The parallel runtime of a program depends on**
  - **input size, processor number, communication/synchronization parameters of the machine**
  - **An algorithm must therefore be analyzed in the context of the underlying platform.**

- **A number of performance measures are intuitive.**
  - **Wall clock time / Completion time**
    - **the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble. But how does this scale when the number of processors is changed or the program is ported to another machine?**
  - **How much faster is the parallel version?**
    - **What's the baseline serial version with which we compare? Can we use a suboptimal serial program to make our parallel program look faster?**
  - **Raw FLOP count**
    - **How good are FLOP counts when they don't solve a problem?**

Overheads in parallel programs

- If I use two processors, shouldn’t my program run twice as fast?

- NO! due to overheads…
  - Communications
  - Interactions
  - Idling due to:
    - Load imbalance
    - Synchronization
    - Serial components
  - Excess computation
    - Sub-optimal algorithms



Performance Metrics

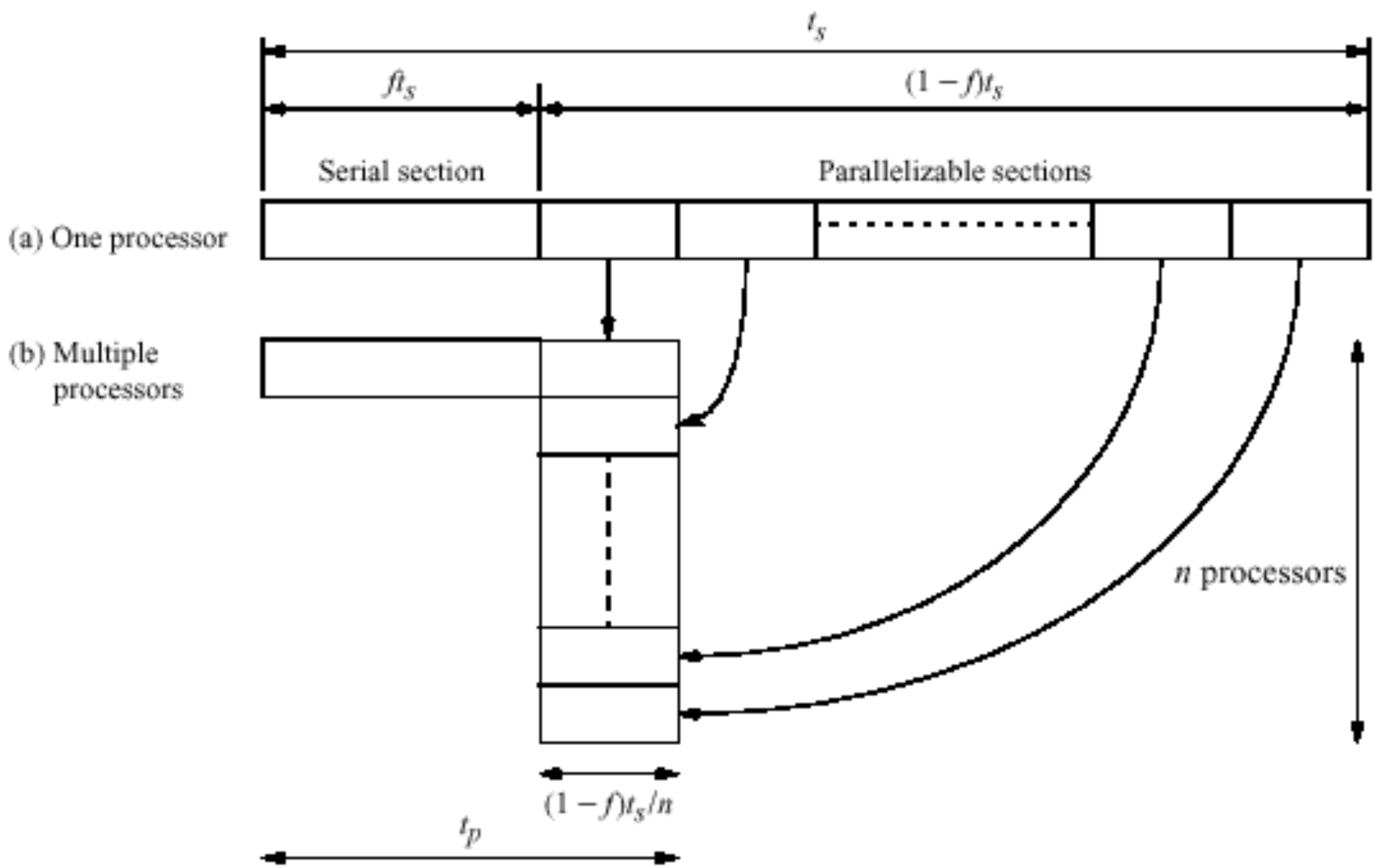
- **Evaluating a parallel algorithm is difficult**
  - **Its performance may also depend on the architecture, on the network, on the homogeneity of the cluster, etc.**
- **Simple measures are:**
  - **Parallel execution time on *n* processors:** $T_P(n)$
  - **Total overhead:** $T_0(n) = n \times T_P(n) - T_S$
    - **Difference between total parallel and serial CPU time, where T_S is the serial time**
  - **Speed-up:** $Sp(n) = T_S / T_P(n)$
    - **How much faster is the parallel version of the algorithm when running on *n* processors?**
  - **Efficiency:** $E(n) = T_S / (T_P(n) \times n) = Sp(n) / n$
    - **Are the parallel resources well exploited?**

Speedup

- **Speedup on *n* processors:** $Sp(n) = T_S / T_P(n)$
- **Linear speedup:** $Sp(n)=n$ **Good!!**
- **Super-linear speedup:** $Sp(n) > n$
  - **due to a sub-optimal sequential algorithm**
  - **due to problem sizes whose data do not fit the memory available on a sequential architecture, while the parallel algorithm is able to exploit the aggregate memory of a parallel architecture**
  - **due to the higher aggregate cache/memory bandwidth of parallel algorithm, which can result in better cache-hit ratios, and therefore super-linearity**
  - **due to due parallel versions that do less work than the corresponding serial algorithm: e.g., *exploratory search***

Amdahl law and Speedup

- **Maximum speedup we can obtain in parallelizing an algorithm**

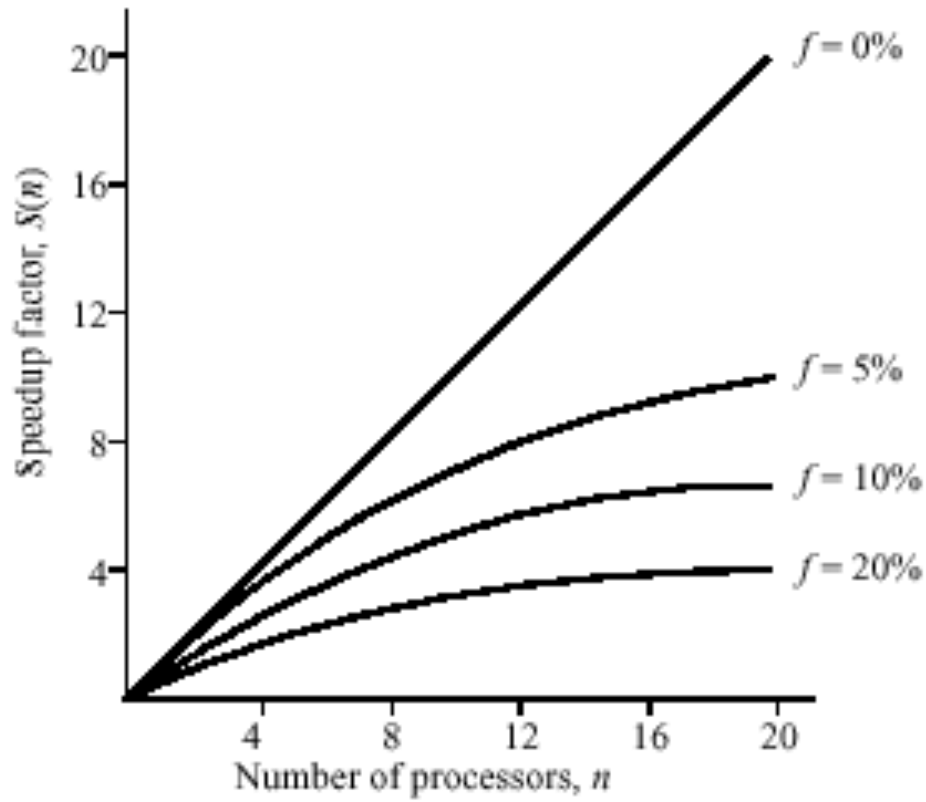


Amdahl law and Speedup

- **Speedup** $$Sp(n) = \frac{t_s}{f\,t_s + (1-f)\,t_s/n}$$

- **If the fraction of code that we cannot parallelize is *f = 1/k*, the Amdahl law states that the maximum speedup we can obtain when *n* → ∞ is:**

  $$Sp(n) = 1/f = k$$



Performance Metrics

- **Cost**: $C(n) = n \times T_P(n) = n \times T_S / Sp(n)$

- ***Cost Optimality***: **A parallel algorithm is cost-optimal if its cost has the same asymptotic growth as T_S, i.e., the fastest serial algorithm (as a function of the input size)**

- **Since efficiency $E(n) = T_S / (T_P(n) \times n)$, for cost optimal systems we have that efficiency is *equal to O*(1).**

Scalability

$$E(n) = Sp(n)/n = T_S / (T_P(n) \times n) = 1 / ((n \times T_P(n) / T_S) =$$
$$= 1/ (T_0(n)/T_S+1)$$

where the total overhead is: $T_0(n) = n \times T_P(n) - T_S$

- **Since $T_0(n)$ increases with n because of its sequential fraction (Amdahl), the efficiency of *any* parallel program *has a reduced efficiency with more processors***

- **In real cases, T_0 increases also for the increased aggregate cost of communication/synchronization, as well as for load imbalance and idling time**

Scalability

$$E(n) = 1/\left(T_0(n)/T_S+1\right)$$

- For many problems/algorithms, *when we increase the problem size*, we observe that T_S grows faster than T_0
  - The parallel part dominates the sequential one
  - The serial fraction does not increase as the problem size
  - Therefore the *efficiency increases*

- For such problems/algorithms it would be possible to keep the efficiency constant when increasing both the number of processors and the problem size.
- Such algorithms are said to be *Scalable*.
- *Scalability* of a parallel system is its ability to increase the speed-up in proportion to the number of processors

Iso-efficiency

- *Question*: what is the most scalable algorithm ?
- *Answer*: Algorithms requiring the problem size to grow at lower rate are more scalable.

- How can we measure such *speed* ?

- Let *W* be the work, i.e. the number of basic computations the best sequential algorithm should executed to solve a given problem.

- We call *W* the *problem size*.
  – Different from the input size.
  – Is a function of the input size.
  – Is actually equal to T_S.

Iso-efficiency

- **From this definition of overhead:**
  - $T_0(W,p) = p\, T_P - T_S = p\, T_P - W$

  we can derive:

  $$T_P = (W + T_0(W,p)) / p$$
- **Speedup is** $S = W / T_P = W\, p / (W + T_0(W,p))$
- **Efficiency is** $E = S/p = W / (W + T_0(W,p))$

  $$= 1 / (1 + T_0(W,p)/W)$$
- **If the problem size is kept constant and** p **is increased:**
  - **the efficiency** E **decreases because the total overhead** $T_0(W,p)$ **increases with** p

Iso-efficiency

- **If *W* is increased keeping *p* fixed, then for scalable parallel systems, the efficiency increases.**
  - **This is because $T_0(W,p)$ grows slower than *W* for a fixed p. For these parallel systems, efficiency can be maintained at a desired value (between 0 and 1) for increasing p, provided W is also increased.**

- **We can derive:**

$$T_0(W,p)/W = (1-E)/E$$

- **The equation tells how much we must increment the problem size *W* wrt the overhead $T_0(W,p)$ – see the ratio on the left - when augmenting the number of processors to keep efficiency constant**